Precise Enforcement of Progress-Sensitive Security

Precise Enforcement of Progress-Sensitive Security

Scott Moore
Harvard University

Aslan Askarov
Harvard University

Stephen Chong
Harvard University

ABSTRACT

Program progress (or termination) is a covert channel that may leak sensitive information. To control information leakage on this channel, semantic definitions of security should be *progress sensitive* and enforcement mechanisms should restrict the channel's capacity. However, most state-of-the-art language-based information-flow mechanisms are progress insensitive—allowing arbitrary information leakage through this channel—and current progress-sensitive enforcement techniques are overly restrictive.

We propose a type system and instrumented semantics that together enforce progress-sensitive security more precisely than existing approaches. Our system is permissive in that it is able to accept programs in which the termination behavior depends only on low-security (e.g., public or trusted) information. Our system is parameterized on a termination oracle, and controls the progress channel precisely, modulo the ability of the oracle to determine the termination behavior of a program based on low-security information. We have instantiated the oracle for a simple imperative language with a logical abstract interpretation that uses an SMT solver to synthesize linear rank functions.

In addition, we extend the system to permit controlled leakage through the progress channel, with the leakage bound by an explicit budget. We empirically analyze progress channels in existing Jif code. Our evaluation suggests that security-critical programs appear to satisfy progress-sensitive security.

Categories and Subject Descriptors

D.4.6 [**Security and protection**]: Information Flow Controls

General Terms

Security, Languages

Keywords

Termination channels, progress channels, information flow

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

1. INTRODUCTION

The security of a system can depend upon the termination behavior of a program. For confidentiality, program termination is a covert channel: if confidential information can influence whether a program terminates or diverges, then an adversary observing program execution may learn this confidential information [4]. For integrity, if untrusted input influences the termination behavior of a program, then an attacker may be able to make a system unavailable, by causing a server loop to exit (e.g., "inputs of death" [12]) or by causing a program to diverge (e.g., "inputs of coma" [13]).

Termination channels have traditionally been considered benign since they were assumed to leak only one bit of information. However, the situation is worse for interactive systems, where an adversarial observer can observe intermediate output [4]. In such systems, it is the *progress* of the computation that leaks secret information. Progress channels may leak an arbitrary amount of information, as illustrated in the program in Listing 1, in which $secret$ is a variable that contains a confidential positive integer. The program prints out to channel L a number of zeroes equal to the value of $secret$, and then diverges. The program thus leaks the value of $secret$ to an observer of channel L.

```
for (i = 0; i < MAXINT; i++) {
    while (i == secret) do skip;
    output_L(0);
}
```

Listing 1: Brute force termination channel

As shown by Askarov et al. [4], a program such as in Listing 1 leaks a 32-bit integer in under 6 seconds. In the presence of concurrency, progress channels have higher bandwidth, because leakage is linear in the number of processes.

Progress-sensitive security [4] is a noninterference-based semantic security condition that prevents information leakage via progress channels. By contrast, *progress-insensitive* security conditions ignore progress channels.

Most strong information security conditions are progress insensitive (e.g., [56, 58, 54, 28, 27, 42]). This is not because progress channels are believed to be benign, but because traditional enforcement mechanisms (such as security-type systems [56, 43] for information-flow control) are not able to precisely control progress channels.

Security-type systems for progress-sensitive security (e.g., [55, 51, 37]) are overly restrictive, disallowing any loops where the loop guard may depend on confidential information. This rules out many useful and secure programs. Indeed, state-of-the-art information flow tools (such as Jif [35], SparkADA [6], and FlowCaml [47])

```
output_L(0);
while (secret > 0) do
  secret := secret − stride;
output_L(1);
```

Listing 2: Progress-sensitive secure if $stride > 0$

are progress insensitive in order to accept useful programs, at the cost of also accepting programs that leak information via progress channels.

We propose a type system and a runtime mechanism that together precisely enforce progress-sensitive security in a simple interactive imperative language. Our system is parameterized on an oracle that reasons about the termination behavior of loops. If the oracle determines that the termination or divergence of a loop depends only on public information, then the termination behavior of the loop does not reveal any confidential information, and execution of the loop may proceed. Otherwise, program execution is terminated, thus preventing leakage of confidential information through the loop's termination behavior. Our system controls the progress channel precisely, modulo the ability of the oracle to determine the termination behavior of loops. As analyses for program termination improve, so will the precision of our enforcement.

The oracle reasons at run time about termination behavior. This allows the oracle to be more precise, as it may use public information specific to a particular execution of the program. It would of course be possible for the oracle to reason statically, providing run-time performance benefits, perhaps at the cost of precision.

Our system soundly enforces progress-sensitive security, rejecting programs that leak information via progress channels. (Such programs would be accepted by existing progress-insensitive type systems.) We have implemented a prototype of the oracle using logical abstract interpretation for termination analysis. This oracle is sufficiently precise to allow us to accept as secure some programs that progress-sensitive type systems reject.

Example. The program in Listing 2 contains a loop with a guard that depends on confidential information: the contents of variable $secret$. An observer of channel L will see a zero output, and, depending on whether the loop terminates, an output of one. Does this program reveal confidential information to the observer of channel L? Provided the public variable $stride$ is positive, the loop is guaranteed to terminate, as the value in $secret$ will eventually be negative. If $stride$ is non-positive, then the termination behavior depends on confidential information: the initial value of $secret$. Since the value of $stride$ cannot be determined statically, a purely static enforcement technique would not be able to accept this program as secure. Provided the oracle is sufficiently sophisticated, in an execution where $stride$ is positive, our system would be able to accept the execution as secure, and allow execution to continue.[1]

```
output_L(0);
while (secret > 0) do
    secret := secret + 1;
output_L(1);
```

Listing 3: Progress-sensitive insecure

[1]Our system actually requires a *cast* annotation on the loop, which indicates that the oracle must examine the loop at run time. The cast annotation is described in Section 2.

Example. By contrast, the program in Listing 3 always reveals confidential information: a value of one is output on channel L if and only if the initial value of $secret$ is non-positive. Most existing information-flow security type systems would accept this program as secure, despite the information leak through the progress channel. Our system rejects this program, since the oracle is unable to prove that the termination behavior of this program depends only on public information.

Budgeted semantics For some programs, it may be acceptable to have some information leaked through progress channels. We extend our system with an explicit *budget* for information leakage through the progress channel. The amount of information leaked via progress channels is tracked at runtime: for each loop that is encountered where the termination behavior may leak high-security information, the budget is reduced. Once the budget limit is reached, the program is terminated, preventing additional confidential information from being leaked. The budget allows us to establish an information theoretic bound on the information leaked via progress channels, and provides a continuum of security between progress sensitive and progress insensitive security conditions.

The rest of the paper is structured as follows. In Section 2 we present a simple interactive imperative language with an annotation that indicates that the termination oracle should be consulted at run time. We present a type system in Section 3. The type system and runtime semantics enforce a progress sensitive security condition, which we discuss in Section 4. Section 5 extends our system and our security guarantees with a budget for leaking information through the progress channel. We extend our results from a simple two-point lattice of security levels to arbitrary security lattices in Section 6.

We have implemented the type system and runtime system, including a termination oracle that uses logical abstract interpretation to reason about program termination and an SMT solver to synthesize linear rank functions to help prove termination. Section 7 describes our implementation. In addition, we have modified the Jif compiler [35] to track progress channels, and applied the modified compiler to a large Jif program, which we also describe in Section 7. By manual inspection we determined that all loops detected by the modified compiler are guaranteed to terminate, and moreover, it is straightforward to reason that they terminate. We thus believe that it is practical and feasible to enforce progress sensitive security.

Section 8 discusses applications and extensions of our system together with a survey of related work. We conclude in Section 9.

2. LANGUAGE AND SEMANTICS

We present a simple imperative language in which to explore enforcement of progress-sensitive security guarantees. We assume a lattice $(\mathcal{L}, \sqsubseteq)$, such that $\mathcal{L}$ is a set of security levels, and $\sqsubseteq$ is a relation over $\mathcal{L}$ that indicates permitted information flow between security levels: for $l_1, l_2 \in \mathcal{L}$, if $l_1 \sqsubseteq l_2$, then information at security level l_1 may flow to security level l_2. We write $l_1 \sqcup l_2$ for the join of l_1 and l_2. For clarity, we initially assume that $\mathcal{L} = \{L, H\}$, and $L \sqsubseteq H$ but $H \not\sqsubseteq L$. Security levels can describe the confidentiality or integrity of information. Intuitively, L represents low security (public or trusted) information, and H represents high security (secret or untrusted) information. In this paper, we focus on reasoning about the confidentiality of information, although our results also apply to integrity. In Section 6 we generalize our results to arbitrary lattices.

Language syntax is presented in Figure 1. Commands are mostly standard, with the exception of an explicit output command, and a

$$
\begin{array}{lrcl}
\text{Values} & v & ::= & n \\
\text{Expressions} & e & ::= & v \mid x \mid e_1 \oplus e_2 \\
\text{Commands} & c & ::= & \mathsf{skip} \mid \mathsf{stop} \mid x := e \mid c_1; c_2 \mid \\
 & & & \mathsf{output}_l(e) \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \\
 & & & \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{cast}_p[c]
\end{array}
$$

Figure 1: Language syntax

"cast" command, described below. Expressions consist of values v, program variables x, and total binary operations over expressions. For simplicity, we restrict values to integers. Output command $\mathsf{output}_l(e)$ evaluates expression e to a value, and outputs the value on channel l. Without loss of generality, we assume that there is one channel for each security level.

We also assume a *security environment* Γ that maps variables to security levels. Intuitively, if $\Gamma(x) = l$, then only information at security level l or below will ever be stored in variable x. The security environment is used both in the runtime semantics of the language (specifically, by the cast command), and in the type system.

Cast command $\mathsf{cast}_p[c]$ dynamically checks whether the termination behavior of c is determined by the current values of low-security variables (i.e., variables x such that $\Gamma(x) = L$). Every command $\mathsf{cast}_p[c]$ has a unique label p, which is used to identify the program point of the command. Whenever this label is clear from the context or is unimportant (as in most of the examples) we omit it for clarity.

Intuitively, we are concerned with protecting the initial values of high-security variables (i.e., variables x such that $\Gamma(x) = H$). We assume that there is an attacker that knows the initial values of low-security variables and observes outputs on the channel for security level L. An execution of a program is regarded as secure if the attacker is unable to learn anything about the initial values of high-security variables. This will be defined formally in Section 4.

Semantics A *memory* is a function from variables to values. We say that two memories m_1 and m_2 are *low equivalent*, written $m_1 \approx_L m_2$, when they agree on the values of low-security variables: $m_1 \approx_L m_2 \triangleq \forall x.\, \Gamma(x) \sqsubseteq L.\, m_1(x) = m_2(x)$.

An *output trace* is a finite list of the form $(v_1, l_1) :: (v_2, l_2) :: \cdots :: (v_n, l_n)$, where each (v_i, l_i) corresponds to an output of value v_i on channel l_i, and (v_n, l_n) is the most recent output. An empty output trace is denoted by ϵ. The *projection* of output trace o, denoted $o \restriction_L$, contains all and only values of o that were output to channel L:

$$
\epsilon \restriction_L = \epsilon
$$

$$
o :: (v, l) \restriction_L = \begin{cases} (o \restriction_L) :: v & \text{if } l \sqsubseteq L \\ o \restriction_L & \text{if } l \not\sqsubseteq L \end{cases}
$$

We say that output traces o_1 and o_2 are *low equivalent*, written $o_1 \approx_L o_2$, if and only if $o_1 \restriction_L = o_2 \restriction_L$.

Program configurations have the form $\langle c, m, o\rangle$ where c is the command to be evaluated, m is the current memory, and o is the output trace produced so far. Semantic transitions have the form $\langle c, m, o\rangle \longrightarrow \langle c', m', o'\rangle$. The transition relation $\longrightarrow$ is mostly standard, and is presented in Figure 2. We write $m(e) = v$ to indicate that expression e evaluates to value v when variables x occurring in e are replaced with their values $m(x)$.

The rule for $\mathsf{cast}_p[c]$ requires that the termination or divergence of command c is determined by low-security information. If that is not the case, then the program is stuck. Rule S-CAST uses a *termination oracle* $\mathsf{O}(p, m, o)$ to determine whether the termination behavior of c depends only on low-security information. The oracle is given a program point p that identifies a cast $\mathsf{cast}_p[c]$, the current memory m, and the current output trace o (and, implicitly, the original program c_0 and the initial memory m_0), and responds with one of TERMINATE, DIVERGE, or UNKNOWN. Intuitively, if the oracle responds TERMINATE then low-security information is sufficient to determine that c will terminate. That is, command c is guaranteed to terminate for any execution of program c_0 that starts with an initial memory that is low equivalent to m_0 and reaches $\mathsf{cast}_p[c]$, after producing an output trace o' that is low equivalent to o, in memory m' that is low-equivalent to m. Formally, if $\mathsf{O}(p, m, o) = \textsc{Terminate}$, then

$$
\begin{array}{l}
\forall m_0', m', o'.\ m_0' \approx_L m_0 \wedge m' \approx_L m \wedge o' \approx_L o. \\
\quad \text{if } (\langle c_0, m_0', \epsilon\rangle \longrightarrow^* \langle \mathsf{cast}_p[c]; c', m', o'\rangle) \\
\quad\quad \text{then there exist } m^* \text{ and } o^* \text{ such that} \\
\quad\quad\quad \langle c, m', o'\rangle \longrightarrow^* \langle \mathsf{stop}, m^*, o^*\rangle.
\end{array}
$$

Similarly, if $\mathsf{O}(p, m, o) = \textsc{Diverge}$ then any low-equivalent execution of c_0 that reaches $\mathsf{cast}_p[c]$ is guaranteed to diverge:

$$
\begin{array}{l}
\forall m_0', m', o'.\ m_0' \approx_L m_0 \wedge m' \approx_L m \wedge o' \approx_L o. \\
\quad \text{if } (\langle c_0, m_0', \epsilon\rangle \longrightarrow^* \langle \mathsf{cast}_p[c]; c', m', o'\rangle) \\
\quad\quad \text{then there does not exist } m^* \text{ and } o^* \text{ such that} \\
\quad\quad\quad \langle c, m', o'\rangle \longrightarrow^* \langle \mathsf{stop}, m^*, o^*\rangle.
\end{array}
$$

S-SKIP
$$
\langle \mathsf{skip}, m, o\rangle \longrightarrow \langle \mathsf{stop}, m, o\rangle
$$

S-ASSIGN
$$
\frac{m(e) = v}{\langle x := e, m, o\rangle \longrightarrow \langle \mathsf{stop}, m[x \mapsto v], o\rangle}
$$

S-SEQ-1
$$
\frac{\langle c_1, m, o\rangle \longrightarrow \langle \mathsf{stop}, m', o'\rangle}{\langle c_1; c_2, m, o\rangle \longrightarrow \langle c_2, m', o'\rangle}
$$

S-SEQ-2
$$
\frac{\langle c_1, m, o\rangle \longrightarrow \langle c_1', m', o'\rangle \qquad c_1' \neq \mathsf{stop}}{\langle c_1; c_2, m, o\rangle \longrightarrow \langle c_1'; c_2, m', o'\rangle}
$$

S-IF
$$
\frac{m(e) \neq 0 \implies i = 1 \qquad m(e) = 0 \implies i = 2}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m, o\rangle \longrightarrow \langle c_i, m, o\rangle}
$$

S-WHILE-TRUE
$$
\frac{m(e) \neq 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ c, m, o\rangle \longrightarrow \langle c; \mathsf{while}\ e\ \mathsf{do}\ c, m, o\rangle}
$$

S-WHILE-FALSE
$$
\frac{m(e) = 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ c, m, o\rangle \longrightarrow \langle \mathsf{stop}, m, o\rangle}
$$

S-OUTPUT
$$
\frac{m(e) = v}{\langle \mathsf{output}_l(e), m, o\rangle \longrightarrow \langle \mathsf{stop}, m, o :: (v, l)\rangle}
$$

S-CAST
$$
\frac{\mathsf{O}(p, m, o) \in \{\textsc{Terminate}, \textsc{Diverge}\}}{\langle \mathsf{cast}_p[c], m, o\rangle \longrightarrow \langle c, m, o\rangle}
$$

Figure 2: Semantics

$$\Gamma, \mathsf{pc} \vdash \mathsf{skip} : L$$

$$\frac{\Gamma \vdash e : l' \qquad \mathsf{pc} \sqcup l' \sqsubseteq \Gamma(x)}{\Gamma, \mathsf{pc} \vdash x := e : L}$$

$$\frac{\Gamma, \mathsf{pc} \vdash c_1 : l_1 \qquad \Gamma, \mathsf{pc} \sqcup l_1 \vdash c_2 : l_2}{\Gamma, \mathsf{pc} \vdash c_1; c_2 : l_1 \sqcup l_2}$$

$$\frac{\Gamma \vdash e : l \qquad \Gamma, \mathsf{pc} \sqcup l \vdash c_i : l_i}{\Gamma, \mathsf{pc} \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : l_1 \sqcup l_2}$$

$$\frac{\Gamma \vdash e : l \qquad \Gamma, \mathsf{pc} \sqcup l \sqcup l' \vdash c : l'}{\Gamma, \mathsf{pc} \vdash \mathsf{while}\ e\ \mathsf{do}\ c : l'}$$

$$\frac{\Gamma, H \vdash c : l}{\Gamma, L \vdash \mathsf{cast}_p[c] : L}$$

$$\frac{\Gamma \vdash e : l' \qquad \mathsf{pc} \sqcup l' \sqsubseteq l}{\Gamma, \mathsf{pc} \vdash \mathsf{output}_l(e) : L}$$

Figure 3: Type system: commands

If the oracle responds with UNKNOWN then the oracle is unable to determine whether the termination behavior of $\mathsf{cast}_p[c]$ depends only on low-security information.

While the problem of proving program termination is clearly undecidable, there are many approaches to implementing sound and useful (but incomplete) termination oracles. Simple, albeit imprecise program analysis could identify common patterns (for example, identifying for loops such that the loop counter is a low-security variable, and the stride is a constant). For more complex programs, existing tools (e.g., [19, 31, 52]) for proving program termination can be employed. More sophisticated approaches, such as the work of Cook et al. [20], are able to automatically synthesize sufficient conditions to prove loop termination. We describe our prototype implementation in Section 7.

3. TYPE SYSTEM

This section presents the typing rules for our language. The rules for expressions are standard and have form $\Gamma \vdash e : l$, meaning that in environment Γ, level l is an upper bound on the information that may be learned by evaluating expression e. Typing rules for commands have form $\Gamma, \mathsf{pc} \vdash c : l$, where pc is the *program counter* level, and l is the termination level. Figure 3 presents typing rules for commands in our language. Termination level l of command c is an upper bound on how much information may be learned by observing c's termination. For simple commands, such as skip, assignment, and output, the termination level is always L—these commands always terminate and thus the termination of the command reveals no information. The rule for sequential composition $c_1; c_2$ propagates the termination level of c_1 into the pc-level of c_2, since c_2 executes only if c_1 terminates. Therefore, if the termination level of c_1 is H, no low assignments or low outputs are allowed in c_2. The termination level of $c_1; c_2$ is the join of the termination levels of the individual commands. The termination level of conditional $\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ is the join of the termination levels of branches c_1 and c_2.

For command $\mathsf{while}\ e\ \mathsf{do}\ c$, the termination of the loop may depend on both the guard expression e and the termination of loop body c. Thus, the termination level for a while loop contains the join of the level of the guard expression l and the termination level of c. In addition, if the while loop diverges then program's nontermination may reveal that the while loop was executed. The pc-level is an upper bound on the information that may be learned by knowing that the while loop executed, and so the pc-level is folded into the termination level of the loop.

Example. Let $\Gamma(h) = H$ and $\Gamma(low) = L$. Consider program

$$\mathsf{while}\ h > 0\ \mathsf{do}\ h := h - low;$$
$$\mathsf{output}_L(1)$$

The type system rejects this program, because the termination level of the while loop is H, and so the pc-level of the output command is also H, which does not type check.

The typing rule for $\mathsf{cast}[c]$ is noteworthy. It ignores the termination level of subcommand c, and the termination level of the casted command is L. The intuition is that the termination behavior of c is *assumed* to depend only on low-security information, and thus the termination or non-termination of c will not leak secret information. This assumption will be validated at runtime by the termination oracle. This is secure because the oracle's decision is based only on low-security information, and thus the success or failure of the cast at runtime does not reveal any secret information. Subcommand c must type check with a pc-level of H, which ensures that c does not contain any assignments to low-security variables or low-security outputs. This is a technical simplification with no loss of expressiveness—any casted command c that is well-typed under low pc-level can be transformed into a form with (possibly many) casts around subcommands of c that are well-typed under high-pc. The latter is possible because a command of the form $\Gamma, L \vdash \mathsf{while}\ h\ \mathsf{do}\ c$ is well-typed if and only if $\Gamma, H \vdash \mathsf{while}\ h\ \mathsf{do}\ c$ is well-typed; the same holds for the conditionals. If original command contains no high loops or conditionals, then the cast is redundant and can be omitted.

Example. Let $\Gamma(h) = H$ and $\Gamma(low) = L$. Consider program

$$\mathsf{cast}[\mathsf{while}\ h > 0\ \mathsf{do}\ h := h - low];$$
$$\mathsf{output}_L(1)$$

This program is accepted by the type system, because the termination level of the cast is L, and so the pc level of the output command is also L, and thus the output command type checks. The termination of the while loop here depends on the values of low and h. In particular, when $low > 0$, the while loop will terminate regardless of the value of h; when $low \leq 0$, termination depends on h. At runtime, according to S-CAST, the termination oracle needs to examine variable low, and execution continues only when $low > 0$.

On placement of casts Note that our language allows casts to be placed around any code block, not just loops. In fact, limiting casts to just loops would be insufficient, as illustrated by the following program.

$$\mathsf{if}\ h > 0$$
$$\quad \mathsf{then}\ \mathsf{while}\ h' > 0\ \mathsf{do}\ h' := h' - low;$$
$$\quad \mathsf{else}\ \mathsf{skip};$$
$$\mathsf{output}_L(1)$$

Placing the cast around the loop would not close the termination channel of this program: when $h \leq 0$, the program takes the else branch, omitting the request to the termination oracle; yet, if $low \leq 0$, the subsequent output of value 1 reveals $h \leq 0$, because otherwise the execution would have been stopped. For this reason, our type system rules out casts in high contexts. The correct placement of cast for this example is around the if command, which would allow the program to type check.

4. SECURITY

We define security in terms of an attacker that is able to observe both the initial values of low-security variables and the low-security output of a program execution. We assume the attacker knows the program text.

We say that configuration $\langle c, m, \epsilon\rangle$ *emits* incomplete trace τ, written $\langle c, m, \epsilon\rangle \downarrow \tau$, if there exists command c', memory m', and output trace o such that $\langle c, m, \epsilon\rangle \longrightarrow^* \langle c', m', o\rangle$ and $o\restriction_L = \tau$. Intuitively, if a configuration emits trace τ then the attacker observes the outputs τ during the execution of the program.

We strengthen the observational model to allow the attacker to determine when an execution will no longer produce additional output on the attacker's channel (because, for example, the program terminates, diverges, or gets stuck). Trace τ is *maximal*, written $\tau\bullet$, when there are no more public outputs possible in the computation that emitted $\tau\bullet$. We say that configuration $\langle c, m, \epsilon\rangle$ *emits* maximal trace $\tau\bullet$, written $\langle c, m, \epsilon\rangle \downarrow \tau\bullet$, if there exists c', m', and o such that $\langle c, m, \epsilon\rangle \longrightarrow^* \langle c', m', o\rangle$ and there is no more public output possible from $\langle c', m', o\rangle$. For example, programs $\mathsf{output}_L(1)$ and $\mathsf{output}_L(1)$; $\mathsf{while}\ 1\ \mathsf{do}\ \mathsf{skip}$ both emit the same maximal trace $(1, L)\bullet$.

We use the term *trace* to refer to both incomplete traces τ and maximal traces $\tau\bullet$, and use metavariable t to range over traces.

Given that an attacker has observed some trace t, the attacker's knowledge [3] is the set of initial memories that could have produced trace t and have the same initial values for all low-security variables. Formally, we have

Definition 1 (Attacker knowledge)**.** Given a program c, initial memory m, and a trace of public outputs t, define *attacker knowledge*

$$k(c, m, t) \triangleq \{m' \mid m \approx_L m' \wedge \langle c, m', \epsilon\rangle \downarrow t\}$$

Attacker knowledge is monotonic in t: the more public outputs are produced in the trace, the fewer memories are consistent with the output. Note that in this definition, a smaller knowledge set corresponds to more precise information.

Our baseline condition for security is *progress-sensitive noninterference* [4].[2]

A program satisfies progress-sensitive noninterference if attacker knowledge remains constant regardless of the observed outputs.

Definition 2 (Progress-sensitive noninterference)**.** Program c satisfies *progress-sensitive noninterference (PSNI)* if for all initial memories m and traces t such that $\langle c, m, \epsilon\rangle \downarrow t$, attacker knowledge does not change, i.e.,

$$k(c, m, t) = \{m' \mid m \approx_L m'\}$$

Example. Program $\mathsf{output}_L(1)$ satisfies Definition 2. The only output produced by this program reveals no secret information to the attacker, and the attacker knowledge is the set of all low-equivalent memories.

However, program $\mathsf{while}\ h > 0\ \mathsf{do}\ \mathsf{skip};\ \mathsf{output}_L(1)$ does not satisfy Definition 2. By Definition 1, we have

$$k(c, m[h \mapsto 0], (1, L)\bullet) = \{m' \mid m'(h) \leq 0 \wedge m \approx_L m'\}.$$

Here $\{m' \mid m'(h) \leq 0 \wedge m \approx_L m'\}$ is the attacker knowledge after seeing output $(1, L)$. Because the attacker knowledge is a strict subset of the set of all memories that are low-equivalent to the initial memory m, the program is insecure.

[2]Askarov et al. [4] call this condition *termination-sensitive* noninterference.

The type system, together with the runtime mechanism, soundly enforces progress-sensitive noninterference. The following theorem is the main result of this section.

Theorem 1 (Soundness of enforcement)**.** *Given program c, if for some security level l we have $\Gamma, L \vdash c : l$ then c satisfies progress-sensitive noninterference.*

The proof of Theorem 1 follows as a special case of Theorem 2, which is presented in the following section.

5. TERMINATION LEAKAGE BUDGET

The operational semantics of Section 2 ensures that if the termination behavior of command $\mathsf{cast}[c]$ cannot be shown to depend purely on low-security information, then the execution will get stuck, preventing any leakage of high-security information. If the execution continued, then the next low-security output produced by the program would allow the attacker to learn that the command terminated, which may reveal high-security information. Similarly, if the command diverges, then the failure to produce another low-security output may allow the attacker to learn high-security information. Indeed, the termination behavior of the program in Listing 1 reveals everything about the initial value of the high-security variable $secret$.

However, continued execution does not necessarily reveal everything about the high-security inputs to the program: the actual information leaked may be less than expected. In this section, we extend the semantics of Section 2 to allow a limited amount of information to be released via the termination behavior of the program.

Consider the following program, in which variables h, h', and $hstep$ are secret ($\Gamma(h) = \Gamma(h') = \Gamma(hstep) = H$), and variable low is public ($\Gamma(low) = L$).

```
1  while low > 0 do {
2      h' := h;
3      cast[
4          while h' > 0 do
5              h' := h' − hstep
6      ]
7      output_L(low);
8      low := low − 1;
9  }
```

Consider the first iteration of the outer loop. The termination or divergence of the inner loop on Lines 4–5 cannot be established using only low-security information. Execution of the low output on Line 7 reveals secret information to the attacker; specifically, it reveals that either $hstep$ is positive or $h \leq 0$.

Now consider the second iteration of the outer loop. Again, the termination or divergence of the inner loop reveals to the attacker the same condition—either $hstep$ is positive or $h \leq 0$. Critically, it does not reveal any more information than was revealed in the first iteration. Indeed, a sufficiently powerful oracle could show that if the program has produced any low output, then any subsequent execution of the inner loop will terminate.

To track and control the amount of information leaked through the termination channel, we introduce a *termination leakage budget* B. The budget bounds the number of outputs that may reveal information about the program's termination behavior.

We extend program configurations to five-tuples $\langle c, m, o, r, s\rangle$, where r is a *release counter* and s is a *pending release bit* that is either 0 or 1. The extended operational semantics will ensure that r counts the number of output events that may allow an attacker

S-CAST-BUDGET

$$\frac{\mathsf{O}(p, m, o) = \textsc{Unknown}}{\langle \mathsf{cast}_p[c], m, o, r, s\rangle \longrightarrow \langle c, m, o, r, 1\rangle}$$

S-OUTPUT-L

$$\frac{m(e) = v \qquad r + s \leq B}{\langle \mathsf{output}_L(e), m, o, r, s\rangle \longrightarrow \langle \mathsf{stop}, m, o :: (v, L), r + s, 0\rangle}$$

S-OUTPUT-H

$$\frac{m(e) = v}{\langle \mathsf{output}_H(e), m, o, r, s\rangle \longrightarrow \langle \mathsf{stop}, m, o :: (v, H), r, s\rangle}$$

Figure 4: Selected rules for budgeted semantics

to learn secret information. Moreover, s will equal 1 only when the production of the next low-security output might reveal secret information to the attacker; when $s = 0$, the next low-security output leaks no information.

5.1 Budgeted semantics

We define a new transition relation for the extended program configurations. We lift every rule in Figure 2 (except for rule S-OUTPUT) to a rule for the new configurations so that r and s remain unchanged. In addition, we add rules in Figure 4.

Rule S-CAST-BUDGET sets s to 1 when the oracle fails to determine whether a cast command will terminate or diverge. Rule S-OUTPUT-L applies when an output to the low-security channel is performed. It increments the release counter by s and clears the pending bit—subsequent public outputs are guaranteed to reveal no information until another cast is reached. Additionally, the rule enforces the termination leakage budget B, requiring that release counter r does not exceed B. Rule S-OUTPUT-H is a lifted rule for outputs on high-security channels. It allows arbitrary output to the high-security channel, and preserves the values of r and s.

The type system for the language does not change.

5.2 Security

Well-typed programs executed using the budgeted semantics may not satisfy progress-sensitive noninterference, but they do satisfy a weaker semantic security condition. To state this condition, we first introduce the notions of *release events* and *progress release events*.

Release events For clarity and easier reference, in the following definitions, we use boxes to highlight relevant semantic transitions. A *release event* is a transition that produces an output that allows the attacker's knowledge to improve: the event releases high-security information to the attacker. As per Theorem 1, well-typed programs executed with the semantics of Section 2 contain no release events.

Definition 3 (Release event)**.** Given a program c, memory m, and an output trace o' such that

$$\langle c, m, \epsilon, 0, 0\rangle \longrightarrow^* \boxed{\langle c', m', o', r', s'\rangle \longrightarrow \langle c'', m'', o'', r'', s''\rangle}$$

then the boxed transition is a *release event* if

$$k(c, m, o' \restriction_L) \supset k(c, m, o'' \restriction_L)$$

Example. In program $low := h; \mathsf{output}_L(low)$, the low output is the release event: attacker knowledge changes with the output.

A particular class of release events that are interesting to us are *progress release events*, which are release events that leak information via the progress channel. That is, it is not the value output, but the fact that the output occurred that reveals information. We capture this intuition by defining progress release events to be release events that reveal only as much information as knowing that *some* output was produced.

Definition 4 (Progress release event)**.** Given a program c, memory m, and an output trace o' such that

$$\langle c, m, \epsilon, 0, 0\rangle \longrightarrow^* \boxed{\langle c', m', o', r', s'\rangle \longrightarrow \langle c'', m'', o'', r'', s''\rangle}$$

then the boxed transition is a *progress release event*, if it is a release event and it holds that

$$k(c, m, o'' \restriction_L) = \bigcup_{v \in \mathbb{Z}} k(c, m, o' \restriction_L :: (v, L)).$$

Here, the term $\bigcup_{v\in\mathbb{Z}} k(c, m, o' \restriction_L :: (v, L))$ is called *progress knowledge* [4].

Example. In program

$$\begin{array}{l}\mathsf{while}\ h > 0\ \mathsf{do\ skip};\\ \mathsf{output}_L(1)\end{array}$$

the low output is a progress release event. We have

$$k(c, m[h \mapsto 0], (1, L)\bullet) = \{m' | m' \approx_L m \wedge m'(h) \leq 0\}.$$

For progress knowledge, we observe that the only possible low output here is exactly $(1, L)$; we have

$$\bigcup_{v\in\mathbb{Z}} k(c, m[h \mapsto v], o' \restriction_L :: (v, L)) = k(c, m[h \mapsto 0], (1, L)).$$

Clearly, by Definition 4 any progress release event is also a release event, but not the other way around. For example, in program $low := h; \mathsf{output}_L(low)$, the low output is not a progress release event.

With the definition of progress release events at hand, we can formulate our theorem for budgeted semantics.

Theorem 2 (Budgeted progress release)**.** *Given a program c such that $\Gamma, L \vdash c : l$ for some security level l then execution of c with budget B contains at most B release events, all of which are progress release events.*

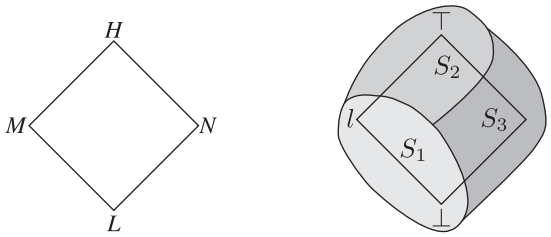
A proof of this theorem is available in the companion technical report [34].

Note that Theorem 1 is a special case of Theorem 2 for budget $B = 0$. Moreover, when B is infinite, Theorem 2 implies that well-typed programs satisfy *progress-insensitive noninterference* [4], a semantic security condition that allows an attacker to learn information through progress release events.

Interpreting the termination leakage budget The budget enforced on a program has a number of interpretations. The simplest interpretation is that B is a bound on the number of times the progress channel may be exploited. An information-theoretic interpretation of this is that the amount of information that may be conveyed via the progress channel is at most $\log_2(B+1)$ bits. This is a pessimistic bound. Because there are at most $B + 1$ possible observations, the expression $\log_2(B + 1)$ bounds both Shannon entropy and min-entropy [49] notions of leakage.

6. MULTI-LEVEL SECURITY

So far we have only considered a simple two-point lattice of security levels. However, many real systems for which information security is a concern require richer lattices to express their security. In this section we extend the language semantics and type system



(a) 4-element Hasse diagram *(b)* Disjoint sets in update function

Figure 5: Example security lattice and disjoint sets in update

to arbitrary security lattices. The extension is non-trivial, and is particularly interesting for budgeted semantics.

Assume an arbitrary lattice of security levels $\mathcal{L}$ with bottom and top elements $\bot$ and $\top$ respectively. We extend the syntax for cast to the form $\mathsf{cast}_p^{l,l'}[c]$ where l and l' are security levels. Level l is an upper bound on the information that the oracle is permitted to use to reason about the termination behavior of command c. In Section 2, this level is implicitly assumed to be L. Level l' is an upper bound on what information is allowed to be leaked by (non)termination of program c. This level is only relevant for budgeted semantics, as explained below, and in Section 5 it is implicitly assumed to be H.

We do not place any restrictions on the relationship between levels l and l'. However, if $l' \sqsubseteq l$ then the termination behavior of the loop is permitted to reveal no more information than the oracle uses to reason about termination behavior, and so the budgeted semantics give no additional benefit over the standard semantics. We thus expect (but do not require) that $l' \not\sqsubseteq l$.

Example. To clarify our exposition throughout this section we use an example four-element security lattice, illustrated by the Hasse diagram in Figure 5a. This lattice contains four security levels L, M, N, H, such that $L \sqsubseteq M \sqsubseteq H$ and $L \sqsubseteq N \sqsubseteq H$, but $M \not\sqsubseteq N$ and $N \not\sqsubseteq M$ (and also $H \not\sqsubseteq M \not\sqsubseteq L$ and $H \not\sqsubseteq N \not\sqsubseteq L$).

Consider the following example program, where variables m, n, and h have security levels M, N, and H respectively.

$$
\begin{array}{l}
h := h + m + n; \\
\mathsf{cast}^{L,H}[\mathsf{while}\ h > 0\ \mathsf{do\ skip};] \\
\mathsf{output}_M(1); \\
\mathsf{output}_L(1);
\end{array}
$$

This program contains a loop that introduces a progress channel: termination of the loop depends on information at levels H, M, and N. The two outputs to levels M and L expose this progress channel. Let us look carefully at what information is revealed by each of these outputs.

The output on M reveals information about H and N to level M. Similarly, output on level L reveals information about M, N, and H to level L. Note that both outputs are potentially dangerous, leaking information to adversaries observing on different channels.

If the order of the two output commands is swapped, as in program

$$
\begin{array}{l}
h := h + m + n; \\
\mathsf{cast}^{L,H}[\mathsf{while}\ h > 0\ \mathsf{do\ skip}]; \\
\mathsf{output}_L(1); \\
\mathsf{output}_M(1);
\end{array}
$$

then we regard only the first output as leaking information. This is because an observer of channel M is also permitted to observe channel L (since $L \sqsubseteq M$). Thus, the first output reveals to level M (and L and N) that the loop terminated, and the second output to level M does not provide any additional information.

6.1 Budgets for multiple levels

We generalize the budgeted semantics by providing a budget for each security level. For example, if each security level represents the information of a security principal, then each principal may set their budget independently. We write $B(l)$ for the leakage budget of security level l. Intuitively, budget $B(l)$ bounds information leakage via progress channels from level l to any other level l' such that $l \not\sqsubseteq l'$.

Extending the release counter and pending release bits We track the number of progress release events for each security level, essentially maintaining a release counter for each security level. Let R be a function from security levels to release counters, such that $R(l)$ is the number of progress release events that have occurred that may have leaked information at level l to some level l' such that $l \not\sqsubseteq l'$. When we encounter an output that can potentially leak information at level l, we conservatively increment the release counters for all levels below l, that is, all l' such that $l' \sqsubseteq l$.

We generalize the pending release bit in a similar way, by tracking a separate pending bit for each security level. Since a function from security levels to a single bit is isomorphic to a set of security levels, we generalize the pending release bit to a set of security levels S. If $l \in S$ then the pending bit for level l is set, meaning that the next output on a channel l', such that $l' \not\sqsubseteq l$, may reveal information from level l. Thus, if $l \in S$ and an output occurs to channel l' such that $l \not\sqsubseteq l'$, then the output release counter for level l is incremented, and l is removed from S. We refer to S as the set of pending levels.

Budget update To formally specify how release counters and the set of pending levels are updated when an output occurs to channel l, we define function $\mathsf{update}(R, S, l)$. This function takes three arguments: R is a release counter, S is a set of pending levels, and l is the security level of the channel on which the output occurred. The function returns a pair (R', S') of the updated release counter R' and the updated set of pending levels S'.

For a fixed security level l, let us rewrite S as a disjoint union of three sets $S = S_1 \uplus S_2 \uplus S_3$ such that

$$
\begin{aligned}
S_1 &= \{l' \mid l' \in S \wedge l' \sqsubseteq l\} \\
S_2 &= \{l' \mid l' \in S \wedge l' \not\sqsubseteq l \wedge l \sqsubseteq l'\} \\
S_3 &= \{l' \mid l' \in S \wedge l' \not\sqsubseteq l \wedge l \not\sqsubseteq l'\}
\end{aligned}
$$

Set S_1 contains levels that flow to l, including l itself. Set S_2 contains all levels that are strictly higher than l, and set S_3 is the set of levels that are incomparable with l. Figure 5b visualizes this partitioning for an arbitrary lattice when S contains all levels.

Example. Consider $S = \{N, H\}$ and $l = M$. Then, according to the definition above, $S_1 = \emptyset$, $S_2 = \{H\}$, and $S_3 = \{N\}$.

Recall that the set S is the set of levels such that an output may reveal information at those levels. After an output on channel l, we need to consume budgets for levels in S_2 and S_3, but not S_1. The budget for levels in S_1 need not be consumed because information at any level $l' \in S_1$ is allowed to flow to level l. Output on channel l may, however, reveal information at levels in S_2 and S_3. We define the updated release counter R' as a function of l' for which it holds that

$$
R'(l') = \begin{cases} R(l') + 1 & \text{if } l' \not\sqsubseteq l \text{ and } \exists l'' \in S_2 \cup S_3.\ l' \sqsubseteq l'' \\ R(l') & \text{otherwise} \end{cases}
$$

This definition increments release counter for all levels l' that do not flow to l, and for which there is a bound l'' in $S_2 \cup S_3$. The condition on the first line of the above definition ensures that we do

Command	S	R			
		L	M	N	H
initial state	$\emptyset$	0	0	0	0
$\mathsf{cast}^{L,H}[\mathsf{while}\ h\ \mathsf{do\ skip}]$	$\{H\}$	0	0	0	0
$\mathsf{output}_M(1)$	$\{M\}$	0	0	1	1
$\mathsf{output}_L(1)$	$\emptyset$	0	1	1	1

Figure 6: Example of budget update in multi-level setting

$$\textsc{S-Cast}\quad \frac{\mathsf{O}(p,m,o) \in \{\textsc{Terminate}, \textsc{Diverge}\}}{\langle \mathsf{cast}_p^{l,l'}[c], m, o, R, S\rangle \longrightarrow \langle c, m, o, R, S\rangle}$$

$$\textsc{S-Cast-Budget}\quad \frac{S' = S \cup \{l'\} \qquad \mathsf{O}(p,m,o) = \textsc{Unknown}}{\langle \mathsf{cast}_p^{l,l'}[c], m, o, R, S\rangle \longrightarrow \langle c, m, o, R, S'\rangle}$$

$$\textsc{S-Output}\quad \frac{m(e) = v \qquad (R', S') = \mathsf{update}(R, S, l) \qquad \forall l'.\ R'(l') \le B(l')}{\langle \mathsf{output}_l(e), m, o, R, S\rangle \longrightarrow \langle \mathsf{stop}, m, o :: (v, l), R', S'\rangle}$$

Figure 7: Budgeted semantics for multi-level setting: selected rules

not unnecessary consume budgets for levels that are not bounded by a level in $S_2 \cup S_3$. For example, when $S_2 \cup S_3$ is an empty set, R does not change.

For the updated pending release bits S', which security levels should be in it? Clearly all of set S_1, as if $l' \in S_1$, a future output at a level l'' such that $l' \not\sqsubseteq l''$ will reveal information about l' via a progress channel. Sets S_2 and S_3 do not need to be in S', as we have already accounted for information leaked via the progress channel for these levels. We may, however, need to add level l to S'. If S_2 is non-empty, then there is a level $l' \in S_2$ such that $l \sqsubseteq l'$. Thus, it is possible that information at level l flowed to level l', where it influenced the termination behavior of the program. Thus, a future output may reveal via a progress channel information at level l that has not yet been accounted for in the budget. We thus define the updated pending release bits S' as follows.

$$S' \triangleq S_1 \cup \{l \mid S_2 \neq \emptyset\}$$

Example. Figure 6 presents an example program together with the set of pending levels and the values of release counters for every level, during this program execution when $h = 0$.

6.2 Semantics

Figure 7 presents budgeted semantics for the extended language. The semantics for the multi-level setting resembles the budgeted semantics of Section 5.1, with the difference that it uses release counter R and the set of pending levels S. Program configurations have the form $\langle c, m, o, R, S\rangle$. As before, the semantics is parametrized over the termination oracle. Generalization of the termination oracle to multi-level setting is straightforward, and we omit it here. The only notable aspect is that when the oracle is given cast label p for cast $\mathsf{cast}_p^{l,l'}[c]$, the oracle is permitted to use only information up to level l to reason about the termination behavior of command c.

As before, rule S-Cast does not modify the release counter R or pending levels S. Rule S-Cast-Budget applies when the oracle returns Unknown. Recall that level l' is an upper bound on the termination level of c. This means that (non)termination of c reveals information up to l'. Subsequently, any output on level l'' such that $l'' \not\sqsubseteq l'$ must consume some part of the termination budget for l'. Therefore, this rule adds l' to the set of pending levels.

Finally, rule S-Output updates the release counter and the pending levels before an output on channel l. Given updated R' and S', the execution is allowed when the budget constraints are satisfied.

6.3 Typing rules

Most of the typing rules from Figure 3 can be extended to the multi-level case in a straightforward manner by replacing any occurrence of level L with level $\bot$. The rule for $\mathsf{cast}_p^{l,l'}[c]$ is more interesting and we show it below.

$$\frac{\Gamma, \mathsf{pc} \sqcup l \vdash c : l'' \qquad l'' \sqsubseteq l'}{\Gamma, \mathsf{pc} \vdash \mathsf{cast}_p^{l,l'}[c] : \mathsf{pc} \sqcup l}$$

The rule requires that c is well-typed with some termination level l''. The only requirement on l'' is that it needs to be bounded by level l'. Because information up to l may be used by the oracle, we require that c is well-typed under context $\mathsf{pc} \sqcup l$. This prevents laundering information through the termination oracle itself. For a similar reason, the termination level of this command is set to $\mathsf{pc} \sqcup l$.

To illustrate this rule, let us consider a few examples.

Example. Program $\mathsf{cast}^{L,M}[\mathsf{while}\ h > 0\ \mathsf{do\ skip}]$ is not well-typed because the termination level of the while loop is H, and $M \not\sqsubseteq H$.

On the other hand, the program $\mathsf{cast}^{L,H}[\mathsf{while}\ h > 0\ \mathsf{do\ skip}]$ is well-typed.

Example. Program

$$\begin{array}{l}\mathsf{cast}^{M,H}[\\ \quad \mathsf{output}_L(1)\\ \quad \mathsf{while}\ h > 0\ \mathsf{do}\ h := h - m\\ ]\end{array}$$

is rightfully rejected by the type system. The release event in this program is subtle. The oracle is allowed to use information up to level M. This means that if $m > 0$, and the oracle can deduce that the while loop will terminate, then the low output preceding the loop will occur. On the other hand, if $m \le 0$, the oracle must return Unknown. The presence of the low output right before the loop will therefore depend on level M, which violates progress-sensitive noninterference.

On nested casts Unlike the simple type system of Section 3, nested casts are allowed in the presence of multiple security levels. The following examples illustrates how such scenario may appear in the presence of budgeted semantics.

Example. Consider the program below which has two nested casts.

$$\begin{array}{l}\mathsf{cast}^{L,M}[\,\mathsf{while}\ m > 0\ \mathsf{do}\ \{\quad \ldots\\ \qquad \mathsf{cast}^{L,H}[\mathsf{while}\ h > 0\ \mathsf{do}\ h{-}{-}] \ldots\\ \quad \}\,]\\ \mathsf{output}_L(1)\,;\end{array}$$

Assume that the termination oracle is unable to prove termination of the outer cast statement, but can prove termination of the inner cast statement. In this case, by the time the execution reaches the output statement, one unit of M leakage budget is consumed, while no leakage budget of H is consumed.

Soundness To formulate soundness for multiple levels we generalize our definitions from Section 4.

First, observe that our definition of projection $o\restriction_l$ from Section 4 easily extends to multiple levels. This allows us to generalize the definition of $\downarrow$ to multiple levels—we write $\langle c, m, o, R, S\rangle \downarrow_l t$

when attacker at level l observes trace t that is produced by configuration $\langle c, m, o, R, S\rangle$. Similarly, we generalize definitions of attacker knowledge at level l, release event at level l, and progress release event at level l.

Note that, in the following definitions, the initial set of pending levels is $\emptyset$, and the initial release counter maps every level to zero.

Definition 5 (Attacker knowledge at level l)**.**

$$k(c, m, t, l) \triangleq \{m' \mid m \sim_l m' \wedge \langle c, m', \epsilon, R_{init}, \emptyset\rangle \downarrow_l t\}$$

where R_{init} is the initial release counter: for all l it holds that $R_{init}(l) = 0$.

Definitions of release event and progress release event, in their turn, use Definition 5.

Definition 6 (Release event)**.** Given a program c, memory m, and an output trace o' such that

$$\langle c, m, \epsilon, R_{init}, \emptyset\rangle \longrightarrow^* \boxed{\langle c', m', o', R', S'\rangle \longrightarrow \langle c'', m'', o'', R'', S''\rangle}$$

then the boxed transition is a *release event* if

$$k(c, m, o' \restriction_l, l) \supset k(c, m, o'' \restriction_l, l)$$

Definition 7 (Progress release event at level l)**.** Given a program c, memory m, and an output trace o' such that

$$\langle c, m, \epsilon, R_{init}, \emptyset\rangle \longrightarrow^* \boxed{\langle c', m', o', R', S'\rangle \longrightarrow \langle c'', m'', o'', R'', S''\rangle}$$

then the boxed transition is a *progress release event*, if it is a release event and it holds that

$$k(c, m, o'' \restriction_l, l) = \bigcup_{v \in \mathbb{Z}} k(c, m, o' \restriction_l, l) :: (v, l).$$

Using these definitions, we can formulate soundness for multiple security levels.

Theorem 3 (Budgeted progress release at l)**.** *Given a program c such that $\Gamma, \bot \vdash c : l'$ for some security level l' then execution of c with budget B contains at most $B(l)$ release events at level l, all of which are release events.*

A proof of this theorem is available in the companion technical report [34].

7. EVALUATION

We have evaluated the feasibility of our approach in two parts. First, we have implemented the (non-budgeted) language semantics and type system for a simple interactive imperative language, including an implementation of a suitable termination oracle. This establishes that techniques for reasoning about program termination can be adapted to reasoning about progress channels. Second, we extended the Jif compiler [35] to track information flow via progress channels, and analyzed a Jif application. We find it is feasible to enforce progress-sensitive security conditions for security-critical applications. We report the details of our evaluation below.

7.1 Prototype implementation

Our prototype termination oracle is based on work by Chawdhary et al. [14], which is a form of logical abstract interpretation over a specialized abstract domain for termination. A particular advantage of this analysis is its performance, compared to analyses that are based on binary reachability (e.g., Terminator [19]). The analysis is parameterized over an algorithm for discovering termination arguments. Following the instantiation given by Chawdhary et al., we use the linear rank synthesis algorithm of Podelski and Rybalchenko [38]. Our termination analysis uses the z3 SMT solver [1] for linear rank synthesis and for eliminating spurious program paths.

Our prototype is furthermore extended with a simple constant propagation analysis that is applied to low variables when casts are encountered at run time. This allows us to find termination arguments that rely on the current run-time values of low variables.

We use our implementation to validate the security of the all examples in this paper. Analyzing a program like the one in Listing 4 results in 31 calls to z3, with an overall time of under 0.8 seconds on a machine with a 2.4 GHz CPU. For more complex programs, this overhead will certainly be larger. Of course, for subprograms that always terminate, like Listing 4, the analysis can be done statically ahead of time. We discuss related work that could improve performance in Section 8.

Because we currently do not take into account the output history of the program, we cannot achieve the tight bound on budget consumption for the example in the beginning of Section 5. An implementation of more precise oracle that would take the output history into account is deferred to future work.

We use our experience with this prototype as a guideline for evaluating the feasibility of enforcing progress-sensitive guarantees in real-world applications, which we discuss next.

```
cast [ while x < 10 do  {
         y := 0
         while y < 10  do  y := y + 1
         x := x + 1  } ]
```

Listing 4: Example program with nested loops where x, y are high variables

7.2 Audit of progress channels in Civitas

Civitas [18] is a remote voting system that provides verifiable results while protecting voter confidentiality. The security of Civitas relies on two factors: strong properties of the underlying cryptographic protocols for voting and information flow guarantees of the implementation. To address the latter, Civitas is implemented in Jif [35], a security-typed language which is believed to enforce a progress-insensitive security condition.

Our premise for this evaluation is that, despite Jif providing only progress-insensitive guarantees, Civitas (and most other security typed programs) satisfy a stronger, progress-sensitive security condition. To evaluate this claim, we extended Jif with our multi-level security type system to track progress channels within methods. We focus only on intra-procedural progress channels, and ignore any inter-procedural progress channels. We identified 66 loops in the Jif standard library and 89 loops in Civitas that require casts to secure possible progress channels. The loops that did not require casts were either dependent on public information or had no low side-effects following them within the containing method. We manually categorized each cast by the termination analysis necessary to demonstrate that it is secure. Figure 8 reports our findings.

Termination analysis Notably, we discovered three simple termination bugs in the Jif standard library. The containsAll method of the AbstractCollection class uses a loop to iterate over elements of the given collection but the loop is missing an increment statement.

	# Loops	# Casts	Termination		
			arith	heap	errors
Jif std-lib	75	66	28	35	3
Civitas	310	89	88	1	-

Figure 8: Audit of casts required to rule out intra-procedural progress channels in the Jif standard library and the Civitas secure voting system. Casts are categorized by the termination argument required to prove them safe: linear arithmetic or heap shape. Loops that are intended to always terminate but may not are reported as errors.

As a result, the method will terminate when called with an empty collection as an argument, but diverge otherwise. In both linked list implementations provided by the library, the hash code of the list is intended to be computed by iterating over this list's elements and combining the hash codes of each, but the current element is not advanced between iterations. Similar to the first bug, these implementations will terminate for empty lists but diverge otherwise. Jif programs using these methods may inadvertently leak information.

All of the remaining loops always terminate, regardless of input. This matches our initial intuition: most programs are intended to terminate and are thus likely to satisfy a stronger, progress-sensitive security condition. Encouragingly, in practice the analysis necessary to prove the absence of progress channels is minimal. We found that all but one of the loops we needed to secure in Civitas were simple loops where the loop counter was a low-security variable, the loop bounds were not changed in the loop body, and the stride was constant. Such loops are easily proven to terminate with existing termination tools, for example by the analysis we use in our prototype implementation. The remaining loop uses a collection iterator from the standard library. Its termination could either be proved directly with a more powerful tool for heap-based termination analysis or by appeal to a model of the standard library.

In the standard library, we found that a heap-based termination analysis, e.g., [8], would be necessary for 35 of the 66 casts. While this type of analysis is more complex, it can be applied once for the library, and subsequent uses of the library can rely on models that express the termination-relevant properties of collections as arithmetic operations [19].

We conclude from this audit that strengthening the guarantees provided by security typed languages is feasible; non-malicious programs are likely to require minimal modification. Thus, it is not unreasonable to require progress-sensitive guarantees from real-world security-critical applications.

8. DISCUSSION AND RELATED WORK

Declassification The budgeted semantics and type system allow the attacker to learn a limited amount of secret information, a form of *declassification*. Much recent work on language-based information flow has considered weakening noninterference using declassification policies to specify *what* information may be released, *when*, *where* and by *whom* (see Sabelfeld and Sands [46] for a survey). Casts for which the oracle is unable to determine whether the command terminates or diverges can be considered a form of *what* information release: the system reveals the termination behavior of the cast. (The information theoretic bound on this information provides a form of quantitative information release, another kind of *what* information release.)

There are several existing security-type systems that enforce declassification policies (e.g., [45, 16]), but the semantic security condition enforced is progress insensitive. We note that even if these type systems were strengthened to enforce a progress-sensitive security condition by rejecting high-security loops (as in, e.g., [32]) it is not clear how declassification annotations may enforce a requirement that secret information is leaked only via the progress channel.

Assume our commands are extended with declassification of expressions, as in e.g., [3], and consider the following program in which the loop guard is explicitly declassified at each iteration.

```
guard := declassify(h > 0)
while guard do {
      h := h − 1;
      guard := declassify(h > 0);
}
```

This program is accepted by the type system of [3]. However, because after the declassification, the loop guard is low, the type system also accepts a program that contains a low output in the body of the loop:

```
1  guard := declassify(h > 0)
2  while guard do {
3        h := h − 1;
4        guard := declassify(h > 0);
5        output_L(1);
6  }
```

This program reveals more information than just the fact that the loop terminates: it reveals the initial value of h if h is positive, similar to the example in Listing 1.

Type systems that are designed to prevent information laundering [44, 2, 7, 33] reject the program above, because of the update to variable h on Line 3, and thus these type systems appear unsuitable for straightforward adaptation to progress sensitivity.

Progress (in)sensitivity Much recent work on language-based information flow relies on progress-insensitive noninterference [4] as the underlying target security condition. Demange and Sands observe [24] that security guarantees of progress-insensitive noninterference may be too weak for small secrets. They distinguish between small and big secrets in programs, and propose a coarse-grained type system that guarantees progress-sensitive security for the small secrets and progress-insensitive security for the big secrets. This approach can easily benefit from the results of our work.

Secure multi-execution [25, 29] addresses the problem of termination channels by enforcing strict isolation between outputs on different security levels. The price is high performance overhead, and non-trivial modification of the semantics of the program. It is moreover unclear whether secure multi-execution may be applied to policies beyond noninterference. Compared to that, our approach is minimally-invasive and does not change the intended semantics of the program. This enables straightforward composition with other work on language-based information flow.

Progress-sensitive enforcement appears in literature on concurrent information flow. The enforcement mechanism of Boudol [10], is, similarly to our approach, parametrized over a class of terminating programs, but unlike our work, it does not take runtime information into account; moreover, nonterminating programs are ruled out. Type systems of [48, 11, 41] enforce progress-sensitivity by permitting high loops but disallowing public side effects after that; this is similar to what one achieves in our language without cast command.

Recent work by Stefan et al. addresses the problem of termination channels by spawning background threads for high computations [53]. A thread may wait upon a spawned computation to inspect its result; doing so reveals whether the computation terminated, and raises the security level of the waiting thread. This technique is largely complimentary to ours, and relies on light-

weight concurrency for efficiency. It may be an adequate alternative to halting program execution when the termination oracle fails or when the progress leakage budget is exhausted.

Integrity While the technical development of this paper focuses on confidentiality, our results apply to integrity as well. Clarkson and Schneider [17] introduce two characterizations of integrity: contamination and suppression. Contamination occurs when untrusted input propagates to trusted output; suppression occurs when the program's output omits correct output. We believe progress integrity attacks can be viewed as a form of suppression.

Termination analysis This work is inspired by recent progress on static analyses for proving termination of realistic imperative programs [19, 31, 52].

As outlined in Section 7, our current prototype implementation uses the logical abstract interpretation for termination analysis of Chawdhary et al. [14]; we currently support programs with linear termination arguments. Because our language semantics are parameterized on an oracle for termination analysis, improvements in automated termination analysis will increase the precision of our enforcement mechanism. In particular, results on proving termination for recursive programs [21] and programs with polynomial [23], bit-vector [22], and heap-based [8] termination arguments offer possibilities for further improving precision of the termination oracle.

Recent work on *conditional termination* [20] statically computes preconditions under which a program terminates. Incorporating these results may lead to more efficient ways to incorporate low-security information at runtime.

Quantitative bounds Our budgeted semantics enforces a simple quantitative bound on the amount of information that may be leaked via a progress channel. Here, our information-theoretic bound of $\log_2(B+1)$ bits of progress leakage is similar in spirit to the bounds presented by Zhang, Askarov, and Myers [5, 57]. A logarithmic bound is also given by Rafnsson and Sabelfeld [40]; they buffer outputs and give the bound in the number of the buffered output batches.

Much recent work on quantitative information flow focuses on what the attacker may learn about the secrets based on a single observation [49, 30]. Incorporating these results provides an interesting avenue for future work.

Smith and Alpizar study non-termination of probabilistic programs [50]. They demonstrate that when probability of nontermination in well-typed programs is small, nontermination does not skew the probability distribution of low outputs. A key technical element of their proof machinery is a program transformation that eliminates all high computations in a program. These results appear particularly relevant for understanding computational security guarantees of programs that use cryptographic primitives (which would otherwise be formulated "modulo termination").

Timing channels Timing channels are known to be a dangerous covert channel in computer systems. Exploiting timing channels requires a strong adversary who has access to an external clock in order to measure timing of the individual outputs. Compared to that attacker model, we assume a weaker but more widespread attacker who is limited to counting the number of low outputs. This attacker model is fairly common in both traditional systems as well as cloud-based batch services, e.g., map/reduce. Because our attacker model considers only a specific aspect of low observations, a more precise characterization of security is possible. Indeed, the information-theoretic bound on the progress channel that we obtain in this work is tighter than the one used in general mitigation of timing channels [5, 57].

Auditing for information flow Work on auditing systems for information flow violations pivots around explicit flow violations (e.g., [36]) or audit of authority decisions for declassification [39, 9, 15]. Our work provides means for auditing progress channel violations. The semantics for low outputs can be augmented to record failed casts without stopping program execution; these records can be subjected to a security audit at a later point in time.

Hybrid type checking Our cast operation is related to Flanagan's hybrid type checking [26], where calls to the oracle in our semantics correspond to dynamic type casts. Unlike hybrid type checking, however, the budgeted semantics permits continuation of computation even if the cast fails, at the cost of consuming a unit of budget.

9. CONCLUSION

We have presented a type system and a runtime mechanism that together precisely enforce progress sensitive information security, controlling information leakage via progress channels (also known as termination channels). The system is parameterized on an oracle that reasons about the termination behavior of loops. We have implemented such an oracle using logical abstract interpretation [14]. To the best of our knowledge, this is the first time cutting-edge static analysis for program termination has been applied to enforce strong information security properties.

We have extended our system to track at run time information leakage via progress channels, and restrict such leakage according to a budget. This provides a continuum of security guarantees between progress-sensitive and progress-insensitive security: a zero budget on leakage enforces progress-sensitive security, and an infinite budget enforces progress-insensitive security.

The paper is primarily concerned with providing security guarantees for confidentiality: we ensure that progress channels do not leak confidential information. However, integrity is a well-known dual of confidentiality, and our results can also provide integrity guarantees, preventing untrusted input from influencing the termination behavior of a program.

Progress-sensitive security provides stronger guarantees than progress-insensitive security. The additional effort required to provide these stronger guarantees appears reasonable. We extended the Jif compiler [35] to track information flow via progress channels, and analyzed Civitas [18], a remote voting system implemented in Jif. The termination behavior of all loops detected by the extended compiler depends only on low-security information and are amenable to existing termination analyses. We conclude that Civitas (and likely other security-typed programs) appears to satisfy a stronger security condition than that implied by the standard Jif type system, and a suitable termination oracle would be able to show this with little additional programmer effort.

Acknowledgements

We thanks the anonymous reviewers for their helpful comments. Andrei Sabelfeld provided helpful comments. This research is supported by the National Science Foundation under Grant No. 1054172 and by the Air Force Research Laboratory.

References

[1] The Z3 Theorem Prover., 2008. Software release, `http://research.microsoft.com/projects/Z3`.

[2] A. Askarov and A. Sabelfeld. Localized delimited release: Combining the what and where dimensions of information re-

lease. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 53–60, June 2007.

[3] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 207–221, May 2007.

[4] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, pages 333–348, Oct. 2008.

[5] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *ACM Conference on Computer and Communications Security*, pages 297–307, 2010.

[6] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, Apr. 2003. ISBN 0321136160.

[7] G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In *Proc. IEEE Computer Security Foundations Symposium*, June 2008.

[8] J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proceedings of the 18th international conference on Computer Aided Verification*, CAV'06, pages 386–400, Berlin, Heidelberg, 2006. Springer-Verlag.

[9] A. Blankstein. Analyzing audit trails in the Aeolus security platform. Master's thesis, MIT, Cambridge, MA, USA, June 2011.

[10] G. Boudol. On typing information flow. In *Proceedings of the Second international conference on Theoretical Aspects of Computing*, ICTAC'05, pages 366–380, Berlin, Heidelberg, 2005. Springer-Verlag.

[11] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[13] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings of the IEEE Computer Security Foundations Symposium*, pages 186–199, Washington, DC, USA, 2009. IEEE Computer Society.

[14] A. Chawdhary, B. Cook, S. Gulwani, M. Sagiv, and H. Yang. Ranking abstractions. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, pages 148–162, 2008.

[15] W. Cheng, D. R. K. Ports, D. Schultz, J. Cowling, V. Popic, A. Blankstein, D. Curtis, L. Shrira, and B. Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 2012. USENIX.

[16] S. Chong and A. C. Myers. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 98–111, Piscataway, NJ, USA, June 2008. IEEE Press.

[17] M. R. Clarkson and F. B. Schneider. Quantification of integrity. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 28–43, Washington, DC, USA, 2010. IEEE Computer Society.

[18] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: Toward a secure voting system. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 354–368, Washington, DC, USA, 2008. IEEE Computer Society.

[19] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 415–426, New York, NY, USA, 2006. ACM.

[20] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 328–340, 2008.

[21] B. Cook, A. Podelski, and A. Rybalchenko. Summarization for termination: no return! *Form. Methods Syst. Des.*, 35(3): 369–387, Dec. 2009.

[22] B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 236–250, 2010.

[23] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin / Heidelberg, 2005.

[24] D. Demange and D. Sands. All Secrets Great and Small. In *Programming Languages and Systems. 18th European Symposium on Programming, ESOP 2009*, number 5502 in LNCS, pages 207–221. Springer Verlag, 2009.

[25] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 109 –124, may 2010.

[26] C. Flanagan. Hybrid type checking. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 245–256, New York, NY, USA, 2006. ACM.

[27] R. Grabowski and L. Beringer. Noninterference with dynamic security domains and policies. In *13th Asian Computing Science Conference, Focusing on Information Security and Privacy*, 2009.

[28] S. Hunt and D. Sands. On flow-sensitive security types. In *Proc. 33rd ACM Symp. on Principles of Programming Languages (POPL)*, pages 79–90, Charleston, South Carolina, USA, Jan. 2006.

[29] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 413 –428, may 2011.

[30] B. Köpf and G. Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *2010 IEEE Computer Security Foundations*, July 2010.

[31] D. Kroening, N. Sharygina, A. Tsitovich, and C. M. Wintersteiger. Termination analysis with compositional transition invariants. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 89–103, Berlin, Heidelberg, 2010. Springer-Verlag.

[32] A. Lux and H. Mantel. Declassification with explicit reference points. In *14th European Symposium on Research in Computer Security*, volume 5789 of *LNCS*, pages 69–85. Springer, 2009.

[33] J. Magazinius, A. Askarov, and A. Sabelfeld. Decentralized delimited release. In *APLAS*, pages 220–237, 2011.

[34] S. Moore, A. Askarov, and S. Chong. Precise enforcement of progress-sensitive security. Technical Report TR-04-12, Harvard School of Engineering and Applied Sciences, 2012.

[35] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.

[36] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[37] K. R. O'Neill, M. R. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 190–201. IEEE Computer Society, June 2006.

[38] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 465–486. Springer Berlin / Heidelberg, 2004.

[39] V. Popic. Audit trails in the Aeolus distributed security platform. Master's thesis, MIT, Cambridge, MA, USA, Sept. 2010. Also as Technical Report MIT-CSAIL-TR-2010-048.

[40] W. Rafnsson and A. Sabelfeld. Limiting information leakage in event-based communication. In *Proceedings of the ACM SIGPLAN Sixth Workshop on Programming Languages and Analysis for Security*, June 2011.

[41] W. Rafnsson, D. Hedin, and A. Sabelfeld. Securing interactive programs. In *Proceedings of the 2012 25th IEEE Computer Security Foundations Symposium*, CSF '12, 2012.

[42] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2010.

[43] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[44] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, Oct. 2004.

[45] A. Sabelfeld and A. C. Myers. A model for delimited release. In *Proc. 2003 International Symposium on Software Security*, number 3233 in Lecture Notes in Computer Science, pages 174–191. Springer-Verlag, 2004.

[46] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 2009.

[47] V. Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[48] G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.

[49] G. Smith. Quantifying information flow using min-entropy. In *QEST*, pages 159–167, 2011.

[50] G. Smith and R. Alpízar. Nontermination and secure information flow. *Mathematical Structures in Computer Science (Special Issue on Programming Language Interference and Dependence)*, 21:1183–1205, Dec. 2011.

[51] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[52] F. Spoto, F. Mesnard, and E. Payet. A termination analyzer for java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3):8:1–8:70, Mar. 2010.

[53] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, June 2012. ACM Press.

[54] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 202–216. IEEE Computer Society, 2006.

[55] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168, 1997.

[56] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3): 167–187, 1996.

[57] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *ACM Conference on Computer and Communications Security*, pages 563–574, 2011.

[58] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd Workshop on Formal Aspects in Security and Trust, IFIP TC1 WG1.7*. Springer, Aug. 2004.